CS 540: Introduction to Artificial Intelligence

Homework Assignment #1

Assigned: Thursday, September 12
Due: Monday, September 23

Hand-in Instructions

This homework assignment includes two written problems and a programming problem in Java. Hand in all parts electronically to your Canvas assignments page. For *each* written question, submit a single **pdf** file containing your solution. Handwritten submissions *must* be scanned. **No photos or other file types allowed**. For the programming question, submit a zip file containing *all* the Java code necessary to run your program, whether you modified provided code or not.

Submit the following **three** files (with exactly these names):

```
<wiscNetID>-HW1-P1.pdf
<wiscNetID>-HW1-P2.pdf
<wiscNetID>-HW1-P3.zip
```

For example, for someone with UW NetID crdyer@wisc.edu the first file name must be: `crdyer-HW1-P1.pdf`

Late Policy

All assignments are due **at 11:59 p.m.** on the due date. One (1) day late, defined as a 24-hour period from the deadline (weekday or weekend), will result in 10% of the total points for the assignment deducted. So, for example, if a 100-point assignment is due on a Wednesday and it is handed in between any time on Thursday, 10 points will be deducted. Two (2) days late, 25% off; three (3) days late, 50% off. No homework can be turned in more than three (3) days late. Written questions and program submission have the same deadline. A total of three (3) free late days may be used throughout the semester without penalty. Assignment grading questions must be discussed with a TA within one week after the assignment is returned.

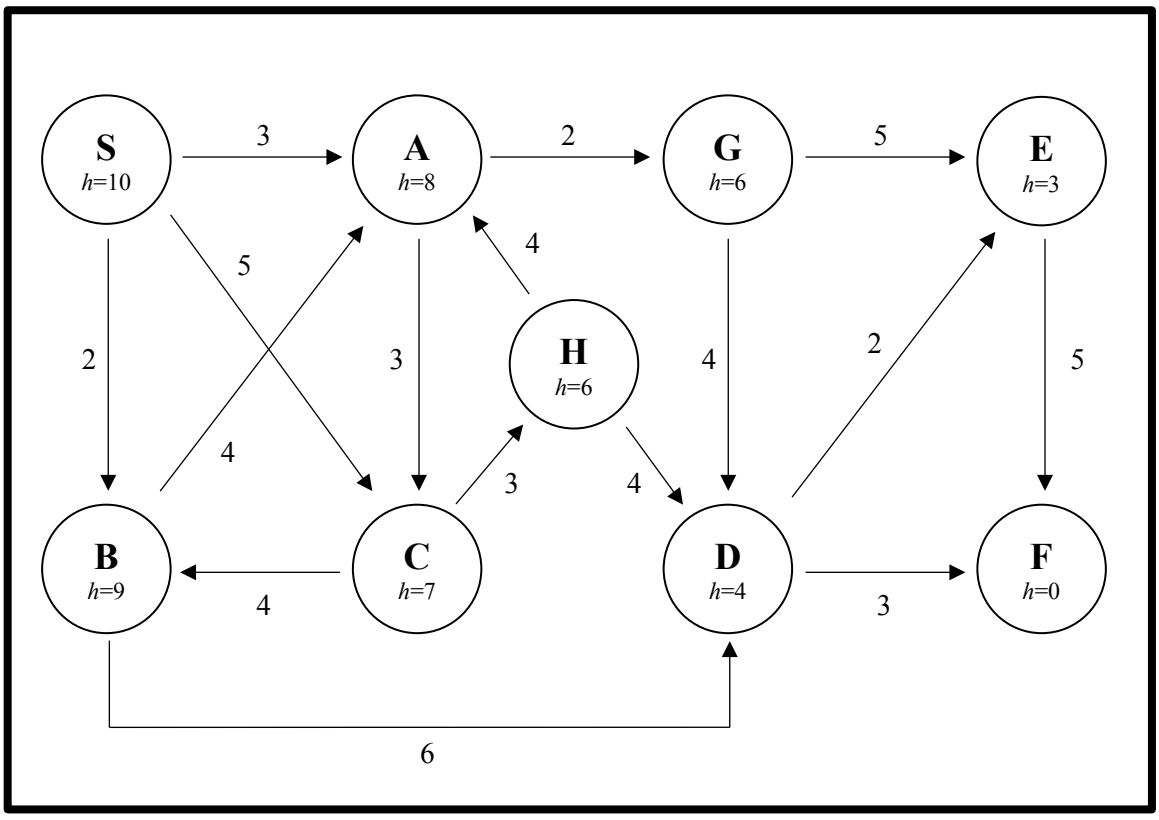
Collaboration Policy

You are to complete this assignment individually. However, you may discuss the general algorithms and ideas with classmates, TAs, peer mentors and instructor in order to help you answer the questions. But we require you to:

- not explicitly tell each other the answers
- not to copy answers or code fragments from anyone or anywhere
- not to allow your answers to be copied
- not to get any code from the Web

Problem 1: Search Algorithms [25 points]

You are given below a state-space graph that consists of **nine** states, the **costs** of the connections between them, and a **heuristic**, ***h*(*n*)**, for each state. Your task is to find a path from start state **S** to goal state **F**. In order to find a solution path, one can use a number of different search methods. In the following questions, you are to find the path from **S** to **F** that the search algorithm given in the question would yield. Use the **tree-search algorithm** given in Figure 3.7 in the textbook where the **goal test** is performed when a state is removed from ***Frontier***. Assume that states are selected/expanded in **alphabetical** order when a **tie** occurs (e.g., if there is a tie between states A and B, then expand A first). **Repeated states** along a path from a node back to the root are **not** allowed. Lastly, if there happen to be several instances of the *same* state in ***Frontier*** when expanding (i.e., two of the same states that have different paths back to **S**), expand first the one that has been in ***Frontier*** longest.



(a) [5] Which solution path will the Depth-First Search (DFS) algorithm find? Expand the successors of a node in alphabetical order (e.g., if a node has 3 successors, A, B, and C, then A will be expanded before B, and B will be expanded before C). Give your answer as one of (i) – (vi) **and show the search tree used to find this solution**.

(i) S – A – C – B – D – E - F
(ii) S – A – C – H – D – F
(iii) S – A – C – B – D – F
(iv) S – B – A – G – D – E – F
(v) S – C – B – A – G – D – F
(vi) DFS will *not* find a solution.

(b) [5] Which solution path will the Breadth-First Search (BFS) algorithm find? Expand the successors of a node in alphabetical order (e.g., if a node has 3 successors, A, B, and C, then A will be expanded before B, and B will be expanded before C). Give your answer as one of (i) – (vi) **and show the search tree used to find this solution**.

(i) S – A – G – E – F
(ii) S – C – H – D - F
(iii) S – B – D – F
(iv) S – A – G – D – F
(v) S – A – G – D – E – F
(vi) BFS will *not* find a solution.

(c) [5] Which solution will Uniform-Cost Search (UCS) find? Give your answer as one of (i) – (vi) **and show the search tree used to find this solution**.

(i) S – B – D – F
(ii) S – C – H – A – G – D – F
(iii) S – B – D – E – F
(iv) S – A – C – H – D – E – F
(v) S – B – A – G – D – E – F
(vi) UCS will *not* find a solution.

(d) [5] Which solution will Greedy Best-First Search find? Give your answer as one of (i) – (vi) **and show the search tree used to find this solution**.

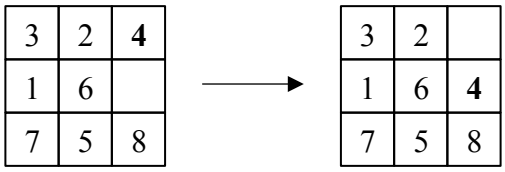
(i) S – A – G – D – F
(ii) S – B – D – E – F
(iii) S – C – H – D – E – F
(iv) S – C – H – D – F
(v) S – C – B – A – G – E – F
(vi) Greedy Best-First Search will *not* find a solution.

(e) [5] Which solution will Algorithm A find? Give your answer as one of (i) – (vi) **and show the search tree used to find this solution**.

(i) S – B – D – F
(ii) S – A – C – B – D – E – F
(iii) S – A – G – D – F
(iv) S – A – C – B – D – F
(v) S – B – D – E – F
(vi) Algorithm A will *not* find a solution.

Problem 2: State-Space Search Trees [10 points]

The **8-puzzle** consists of eight numbered tiles on a 3 x 3 board. The object is to go from a starting state to a goal state by sliding tiles horizontally or vertically (**not diagonally**) using the empty space. For this problem, **assume** that if a state has been reached previously along the path back to the root in the search tree, you **cannot** go back to that state again (i.e., repeated state checking is done to avoid loopy paths).

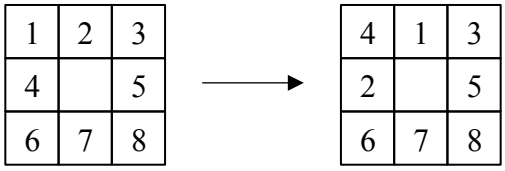


An example move in the 8-puzzle.

(a) [3] From some state in the 8-puzzle, what can be the ***minimum*** number of possible moves (i.e., the minimum number of legal successors)?

(b) [3] From some state in the 8-puzzle, what can be the ***maximum*** number of possible moves (i.e., the maximum number of legal successors)?

(c) [4] What is the ***minimum*** number of moves needed to reach the goal state given below? Justify your answer by drawing a portion of the search tree that proves this.



Initial state Goal state

Problem 3: Maze Solver [65 points]

Given a two-dimensional maze with a starting position and a goal position, your task is to write a Java program called `FindPath.java` that assists a robot in solving the maze by finding a path from the start node to the goal node (as described below) if one exists. The program must accept command line arguments in the following format:

```
$ java FindPath maze search-method
```

The first argument, `maze`, is the path to a text file containing the input maze as described below, and the second argument, `search-method`, can be either "`bfs`" or "`astar`" indicating whether the search method to be used is breadth-first search (BFS) or A* search, respectively.

The Maze

A maze will be given in a text file as a matrix in which the start position is indicated by "`S`", the goal position is indicated by "`G`", walls are indicated by "`%`", and empty positions where the robot can move are indicated by " ". The outer border of the maze, i.e., the entire first row, last row, first column and last column will *always* contain "`%`" characters. A robot is allowed to move only horizontally or vertically, not diagonally.

The Algorithms

For both BFS and A* search, explore the surrounding positions in the following order: move-Left (L), move-Down (D), move-Right (R), and move-Up (U). In BFS, add the successors in that order to the queue that implements the *Frontier* set for this search method. In this way, moves will be visited in the same order as insertion, i.e., L, D, R, U. Assume all moves have cost 1. Repeated state checking should be done by maintaining both *Frontier* and *Explored* sets. If a newly generated node, *n*, does *not* have the same state as any node already in *Frontier* or *Explored*, then add *n* to *Frontier*; otherwise, throw away node *n*.

For A* search, use the heuristic function, *h*, defined as the Euclidean distance from the current position to the goal position. That is, if the current (row#, column#) position is (*u*, *v*) and the goal position is (*p*, *q*), the Euclidean distance is $\sqrt{(u-p)^2+(v-q)^2}$. Add moves in the order L, D, R, U to the priority queue that implements the *Frontier* set for A* search. Assume all moves have cost 1. For A* search, repeated state checking should be done by maintaining both *Frontier* and *Explored* sets as described in the Graph-Search algorithm in Figure 3.14 in the textbook. That is,

- If a newly generated node, *n*, does *not* have the same state as any node already in *Frontier* or *Explored*, then add *n* to *Frontier*.
- If a newly generated node, *n*, has the *same* state as another node, *m*, that is already in *Frontier*, you must compare the *g* values of *n* and *m*:
  - If $g(n) \geq g(m)$, then throw node *n* away (i.e., do *not* put it on *Frontier*).
  - If $g(n) < g(m)$, then remove *m* from *Frontier* and insert *n* in *Frontier*.
- If new node, *n*, has the *same* state as previous node, *m*, that is in *Explored*, then, because our heuristic function, *h*, is consistent (aka monotonic), we know that the optimal path to the state is guaranteed to have already been found; therefore, node *n* can be thrown away. So, in the provided code, *Explored* is implemented as a Boolean array indicating whether or not each square has been expanded or not, and the *g* values for expanded nodes are not stored.

Output

After a solution is found, print out on separate lines:

1. the maze with a "." in each square that is part of the solution path
2. the length of the solution path
3. the number of nodes expanded (i.e., the number of nodes removed from *Frontier*, including the goal node)
4. the maximum depth searched
5. the maximum size of *Frontier* at any point during the search.

If the goal position is *not* reachable from the start position, the standard output should contain the line "`No Solution`" and nothing else.

Code

You must use the code skeleton provided. You are to complete the code by implementing `search()` methods in the `AStarSearcher` and `BreadthFirstSearcher` classes and the `getSuccessors()` method of the `State` class. **You are permitted to add or modify the classes and methods, but we require you to keep the IO class as is for automatic grading.** The `FindPath` class contains the main function.

Compile and run your code using an IDE such as Eclipse. To use Eclipse, first create a new, empty Java project and then do `File` → `Import` → `File System` to import all of the supplied java files into your project.

You can also compile and run with the following commands in a terminal window:

```
javac *.java
java FindPath input.txt bfs
```

Testing

Test both of your search algorithms on the sample test input file: `input.txt` and compare your results with the two output files: `output_astar.txt` and `output_bfs.txt`. Make sure the results are correct on CSL machines.

Deliverables

Put *all* `.java` files needed to run your program, including ones you wrote, modified or were given and are unchanged, into a folder called `<wiscNetID>-HW1-P3` Compress this folder to create `<wiscNetID>-HW1-P3.zip` and upload it to Canvas. For example, for someone with UW NetID crdyer@wisc.edu the file name must be: `crdyer-HW1-P3.zip`